Design Patterns

The design patterns explored in this chapter are as follows:

• Factory

• Proxy

• Decorator

• Adapter

• Strategy

• State

• Template

• Middleware

• Command

**Factory**

First and foremost, a factory allows us to separate the object creation from its

implementation; essentially, a factory wraps the creation of a new instance, giving

us more flexibility and control in the way we do it. Inside the factory, we can create

a new instance leveraging closures, using a prototype and the new operator, using

Object.create(), or even returning a different instance based on a particular

condition. The consumer of the factory is totally agnostic about how the creation of

the instance is carried out. The truth is that, by using new, we are binding our code

to one specific way of creating an object, while in JavaScript we can have much more

flexibility, almost for free. As a quick example, let's consider a simple factory that

creates an Image object:

function createImage(name) {

return new Image(name);

}

var image = createImage('photo.jpeg');

var image = new Image(name);// javascript way

**A mechanism to enforce encapsulation**

A factory can also be used as an *encapsulation* mechanism, thanks to closures.

*Encapsulation* refers to the technique of controlling the access to

some internal details of an object by preventing the *external* code

from manipulating them directly. The interaction with the object

happens only through its public interface, isolating the external code

from the changes in the implementation details of the object. This

practice is also referred to as *information hiding*. Encapsulation is also

a fundamental principle of object-oriented design, together with

inheritance, polymorphism, and abstraction.

**Proxy**

A **proxy** is an object that controls the access to another object called **subject**.

The proxy and the subject have an identical interface and this allows us to

transparently swap one for the other; in fact, the alternative name for this

pattern is *surrogate*. A proxy intercepts all or some of the operations that are

meant to be executed on the subject, augmenting or complementing their

behavior. The following figure shows the diagrammatic representation:



The preceding figure shows us how the **Proxy** and the **Subject** have the same

interface and how this is totally transparent to the client, who can use one or the

other interchangeably. The Proxy forwards each operation to the subject, enhancing

It's important to observe that we are not talking about proxying

between classes; the Proxy pattern involves wrapping actual

instances of the subject, thus preserving its state.

**Data validation**: The proxy validates the input before forwarding it

to the subject

• **Security**: The proxy verifies that the client is authorized to perform the

operation and it passes the request to the subject only if the outcome of

the check is positive

• **Caching**: The proxy keeps an internal cache so that the operations are

executed on the subject only if the data is not yet present in the cache

• **Lazy initialization**: If the creation of the subject is expensive, the proxy

can delay it to when it's really necessary

• **Logging**: The proxy intercepts the method invocations and the relative

parameters, recoding them as they happen

• **Remote objects**: A proxy can take an object that is located remotely,

and make it appear local

**Decorator**

**Decorator** is a structural pattern that consists of dynamically augmenting the

behavior of an existing object. It's different from classical inheritance, because the

behavior is not added to all the objects of the same class but only to the instances

that are explicitly decorated.



Implementation-wise, it is very similar to the Proxy pattern, but instead of enhancing

or modifying the behavior of the existing interface of an object, it augments it with

new functionalities, as described in the following figure:

In the previous figure, the Decorator object is extending the Component object by

adding the methodC() operation. The existing methods are usually delegated to the

decorated object, without further processing. Of course, if necessary we can easily

combine the Proxy pattern, so that also the calls to the existing methods can be

intercepted and manipulated.

Decorator-levelup-plugin code:

export function levelSubscribe (db) {

    db.subscribe = (pattern, listener) => {

        db.on('put', (key, val) => {

            const match = Object.keys(pattern).every(

                k => (pattern[k] === val[k])

            )

            if (match) {

                listener(key, val)

            }

        })

    }

    return db

}

1. **export function levelSubscribe(db) {**: This line defines and exports a function named **levelSubscribe** that takes a parameter **db**.
2. **db.subscribe = (pattern, listener) => {**: Adds a new method **subscribe** to the **db** object. This method takes two parameters: **pattern** (an object) and **listener** (a function).
3. **db.on('put', (key, val) => {**: Listens for the 'put' event on the **db** object. The 'put' event typically occurs when a new key-value pair is added to the database.
4. **const match = Object.keys(pattern).every(k => (pattern[k] === val[k]));**: Checks if every key in the **pattern** object matches the corresponding value in the incoming **val** object. It uses the **every** method to iterate over the keys and compares them with the values in the **val** object.
5. **if (match) { listener(key, val); }**: If all key-value pairs in the **pattern** match the incoming **val**, it calls the **listener** function with the **key** and **val** parameters.
6. **})**: Closes the 'put' event listener.
7. **}**: Closes the **subscribe** method.
8. **return db;**: Returns the modified **db** object, now with the added **subscribe** method.

In summary, this code extends a given database (**db**) by adding a **subscribe** method. This method allows you to listen for 'put' events and filter them based on a specified **pattern**. If a new key-value pair matches the pattern, the provided **listener** function is called with the key and value.